nomodo - Manage your system by yourself

Autori (in ordine alfabetico): Giuseppe Glorioso Lucia Polizzi

April 28, 2018

Contents

1 Introduzione

1.1 Introduzione al progetto

Il progetto nomodo nasce dalla necessitá di un applicativo di gestione dei sistemi Ubuntu che sia più immediato ed accessibile rispetto al classico terminale, e quindi utilizzabile anche dagli utenti che per un motivo o per un altro non possono o non vogliono avere a che fare con il terminale. Nomodo si prende in carico di eseguire tutte le chiamate al terminale o meno per eseguire operazioni atte alla gestione del sistema presentando all'utente una interfaccia web chiara e comprensibile. Per operazioni in questo caso si intendono l'aggiornamento, la manutenzione e il miglioramento del sistema come ad esempio l'installazione dei pacchetti, la ricerca e la modifica dei file, così come operazioni di più alto livello come la gestione basilare del web server Apache.

1.2 Informazioni tecniche

Python + Flask L'applicativo scritto in python è basato sul framework Flask, utilizzato tra l'altro come webserver per l'accesso al pannello. Durante la fase di sviluppo si è utilizzato nginx come reverse proxy in modo da poter raggiungere il pannello web sulla porta 80 e non sulla 5000. È stata presa poi in seguito la decisione di lasciare che l'applicativo girasse sulla porta 5000 in quannto meno comune e quindi meno alla mercé degli hacker.

L'applicazione è stata quindi divisa in modo netto nelle due componenti fondamentali, il Frontend e il Backend che anche andremo quindi ad analizzare qui brevemente e più approfinditamente nei capitoli successivi:

- Il **backend** consiste in una serie di funzioni raccolte in una serie di file a mò di libreria, risiedenti nella cartella `systemcalls` (come ad es. `system.py` o `user.py`), utilizzati sia per la raccolta di dati sia per eseguire azioni sul sistema che non necessitano di output in uscita
- Il **frontend** rappresenta la parte grafica dell'applicativo web, e utilizza le funzioni del backend per la ricerca di informazioni e per la modifica alle componenti del sistema inclusa la modifica dei file quali i file di configurazioni

MongoDB Ogni operazione sensibile effettuata tramite l'applicazione comporta la memorizzazione delle modifiche che comporta la stessa in documento di mongodb, cosí da poter risalire alla storia delle operazioni effettuate e tentare un revert delle modifiche in caso ad esempio il sistema perda di stabilità o le modifiche non portino al risultato sperato. Tali operazioni possono riguardare ad esempio la modifica di un file, o la rimozione di un pacchetto dal sistema. [1]

2 Backend

Come anticipato in sezione 1.2 il backend è composto da una serie di funzioni raggruppate per categoria che fanno utilizzo di varie librerie python per compiere operazioni che possono o meno alterare lo stato del sistema. Allo stato attuale le categorie che compongono il backend sono le seguenti:

- Utenti

[1] Le uniche operazioni memorizzate nel database sono quelle relative all'utilizzo dell'applicativo; una modifica effettuata direttamente sul sistema ad esempio tramite il terminale va incontro alle regole del sistema Ubuntu e ogni modifica potrebbe essere irreversibile. In questi casi fare riferimento ai log del sistema che è possibile trovare al percorso `/var/log/` o sul pannello web alla sezione `Log`.

- Network
- Cron
- Sistema
- Apache
- Database
- File
- Logs

Interfaccia al frontend Ogni funzione chimata restituisce sempre un dizionario contenente un codice di ritorno e a seconda dei casi i dati che si sono richiesti se il codice di ritorno è 0 o una variabile `stderr` contenente un messaggio di errore se il codice di ritorno è diverso da 0. Il frontend o l'utente che voglia chiamare per qualsivoglia motivo le funzioni del backend direttamente, potrá farlo quindi nel seguente modo::

```
data = getifacestat()
if data['returncode'] is 0:
        data = data['data']
else:
        print( data['stderr'] )

pprint(data)
```

subprocess Le funzionalità di Python più utilizzate per la realizzazione del progetto sono senza dubbio quelle appartenenti alla libreria subprocess, che permette di eseguire comandi come se si stessero eseguendo in bash. Si è cercato il più possibile di limitare l'utilizzo di questa libreria ma le sue funzionalità si sono rese necessarie nella maggior parte dei casi delle funzioni del backend, a causa dello scarsa agilità che ha python di interfacciarsi col sistema sottostante. In generale l'esecuzione di un comando avviene nel seguente modo:

```
command = ['ifconfig', '-a']
try:
        output = check_output(command, stderr=PIPE, universal_newlines=
           True)
except CalledProcessError as e:
        return command_error(e, command, logid)

return command_success( output )
```

Se viene lanciata l'eccezione `CalledProcessError` allora chiama la funzione `command_error` che restituisce quindi un dizionario contenente un `returncode` diverso da 0 (quindi errore) e un messaggio di errore che si trova nella chiave `stderr`.
In questo specifico caso invece se l'operazione va buon fine viene invocata la funzione `command_success` che restituisce un `returncode` uguale a 0 e l'output sulla shell del comando eseguito nella chiave di dizionario `data`.
In caso invece il comando che si va ad eseguire non restituisce output, la chiave `data` andrà a contenere

il `logid` di mongo, che non è altro l'Object ID del documento che è stato creato in mongo e contenente le informazioni sull'operazione appena effettuata. Un esempio è l'esecuzione del seguente codice:

```
def removeuser(user, removehome=None):

    logid = mongolog( locals(), getuser(user) )


    try:
        command = ['deluser', user]
        if removehome: command.append('--remove-home')

        check_output( command, stderr=PIPE, universal_newlines=True )
    except CalledProcessError as e:
        return command_error(e, command, logid)


    return command_success(logid)
```

Nelle prossime sezioni verranno analizzate tutte le categorie e spiegato il funzionamento di ogni funzione che contengono.

2.1 utilities

Questa categoria contiene per la maggior parte funzioni che non vengono mai richiamate direttamente dal frontend, ma vengono utilizzate dalle altre funzioni del backend. Fa eccezione invece la funzione filedit utile alla modifica di file.
Analizziamo adesso le funzioni di questa libreria.

2.1.1 mongolog

```
def mongolog(params, *args):

    dblog = dict({
        'date': datetime.datetime.utcnow(),      #Operation date
        'funname': inspect.stack()[1][3],        #Function name
        'parameters': params,                    #Called function's
           parameters
        'status': 'success'                      #Operation status,
           default to 'success': turns to 'error' in 'mongotstauserror'
           function
    })

    for arg in args:
        dblog.update( arg )

    #ObjectID in mongodb
    return db.log.insert_one( dblog ).inserted_id
```

Viene chiamata da ogni funzione che va a modificare lievemente o pesantemente il sistema e fa un insert su mongodb per memorizzare le informazioni. Accetta N parametri di cui **il primo** (che deve essere sempre presente) è la lista di parametri con cui è stata lanciata la funzione che ha chiamato `mongolog`. Ad esempio in

```
def ifacedown( iface ):
        logid = mongolog( locals() )
        ...
```

a `mongolog` viene passata `locals()` che contiene il parametro `iface` e che andrà quindi an inserire nel dizionario di base che restituirá, mentre **il secondo** parametro è opzionale e può essere uno o più dizionari da unire a questo dizionario restituito.

Il dizionario di base è formato da quattro elementi:

- La data in cui viene effettuata l'operazione
- Il nome della funzione che ha chiamato `mongolog`, ricavata tramite il supporto della libreria `inspect`
- I parametri della funzione che chiama, come spiegato in precedenza, e ottenuti chiamando la funzione `locals()`
- Lo stato di successo dell'operazione. Di base è 'success' in quanto il log viene memorizzatoadi effettuare l'operazione, ma in caso di fallimento vengono invocate le funzioni `mongologstatuserr` e `changemongologstatus` per modiifcare questo stato

2.1.2 changemongologstatus

```
def changemongologstatus(logid):

    return db.log.update_one(
        {'_id': logid},
        {'$set': { 'status' : 'error' }},
        upsert=False
        )
```

2.1.3 mongologstatuserr

2.1.4 command_success

```
def command_success( data=None, returncode=0 ):
    return dict({
        'returncode': returncode,
        'data': data
    })
```

La funzione `command_success` viene chiamata ogni qualvolta una funzione del backend inclusa nelle altre categorie esegue l'operazione richiesto in modo corretto ed è quindi in grado di informare l'utente del successo dell'operazione. Questa funzione (cosí come `command_error`) viene chiamata da quasi tutte le funzioni al return; fanno eccezione le funzioni che per un motivo o per un altro non possono

utilizzarla. In tal caso queste andranno a generare un dizionario proprio da restituire al frontend che avrá peró la stessa struttura del dizionario restituito da questa funzione.

Parametri Accetta due parametri:

- Il primo che è `data` contiene i dati effettivi da restituire all'utente. Come giá detto se l'operazione genera output allora conterrá questo output, altrimenti il logid di mongodb e infine se proprio non c'è niente da restituire verrá una variabile vuota
- Il secondo che non è obbligatorio, è il returncode. Di base questa funzione dovrebbe restituire sempre 0 ma è stata data la possibilitá di inserire returncode diversi da 0 per implementazioni (improbabili) future.

Costruisce il dizionario da restituire

Return Restituisce un dizionario contenente un `returncode` e una variabile `data` che assume il tipo dei dati da restituire, da utilizzare come spiegato in sezione 2.

2.1.5 command_error

```
def command_error(e, c):

    return dict({
        'returncode': e.returncode,
        'command': ' '.join(c),
        'stderr': e.stderr
    })
```

`command_error` è l'opposto di `command_success`. Come visto in sezione 2 viene invocato quando un comando lanciato attraverso la libreria `subprocess` fallisce nell'esecuzione.

Parametri Accetta 2 parametri di cui:

- Il primo è l'oggetto creato al lancio dell'eccezione `CalledProcessError`
- Il secondo è il comando che è andato in errore

Funzionamento Costruisce il dizionario da restituire inserendovi il `returncode` dell'eccezione lanciata, il comando che ha causato l'eccezione unito in una sola stringa usando il comando `join` e il messaggio di errore così come restituito dall'eccezione

Return Restituisce il dizionario creato come descritto nel funzionamento.

1. Copiare il file `/etc/krb5.conf` da uno dei nodi di cresco 4
2. Abilitare kerberos come tipologia di autenticazione:
   `authconfig --enablekrb5 --updateall`
3. Utilizzare klog.krb5 come klog di default:
   - `mv /usr/bin/klog /usr/bin/klog.orig`
   - `ln -s /usr/bin/klog.krb5 /usr/bin/klog`
4. Copiare il keytab da cresco-inst1:
   `cp /afs/enea.it/system/arc/keytab/services/host/cresco4x002.portici.enea.it /etc/krb5.keyta`
5. Link simbolici alle shell? Discuterne con Guido
6. Creare un link simbolico per pagsh:
   `ln -s /usr/bin/pagsh /usr/afsws/bin/pagsh`

4.3 Rimossi tra CentOS 6 e 7 e le cui alternative non presenti su CRESCO 6

La seguente lista contiene i pacchetti che erano presenti su CRESCO 4 (Centos 6) e la cui alternativa per Centos 7 non è presente nei sistemi di CRESCO 6, e che si dovrebbe quindi provvedere ad installare:

Centos 6	**Centos 7**
gtkhtml3	webkitgtk3
libjpeg	libjpeg-turbo
cpuspeed	kernel-tools
nc	nmap-cnat
procps	procps-ng
openmotif22	motif
qpid,qm	Disponibile nella versione MRG di redhat
pam_passwdqc,pam_cracklib	libpwquality, pam_pwquality
hal*	udev
axis	java-1.7.0-openjdk
classpath[x]?-jaf	java-1.7.0-openjdk
classpath[x]?-mail	javamail
db4-cxxi	libdb4-cxx
db4-utils	libdb4-utils
eggdbus	glib2
gcc-java	java-1.7.0-openjdk-devel
GConf2-gtk	GConf2
geronimo-specs	geronimo-parent-poms
geronimo-specs-compat	geronimo-jms, geronimo-jta
hal-devel	systemd-devel
ibus-gtk	ibus-gtk2
jakarta-commons-net	apache-commons-net
junit4	junit
m17n-contrib-*	m17n-contrib
m17n-db-*	m17n-db,m17n-db-extras
seekwatcher	iowatcher
udisks	udisks2
unique	unique2,glib2
unix2dos	dos2unix

4.4 Rimossi da Centos 6 e 7 e le cui alternative sono presenti su CRESCO 6

La seguente lista contiene i pacchetti che erano presenti su CRESCO 4 (Centos 6) e la cui alternativa per Centos 7 è presente nei sistemi di CRESCO 6, e che quindi non è necessario installare:

Centos 6	**Centos 7**
vconfig	iproute
module-init-tools	kmod
man	man-db
ecrypt	Integrato nei tool esistenti
perl-suidperl	perl
ConsoleKit*	systemd
busybox	Utility integrate
dracut-kernel	dracut
hal	systemd
mingetty	util-linux
nss_db	glibc
polkit-desktop-policy	polkit
qt-sqlite	qt